The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 17, No. 3, pp.1-7, Jun. 30, 2017. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2017.17.3.1
JIIBC 2017-3-1

큐를 이용한 이산대수의 사이클 검출

Cycle Detection in Discrete Logarithm Using a Queue

이상운*

Sang-Un, Lee*

요 약 본 논문은 $\alpha^{\gamma} \equiv \beta (\mathrm{mod}\, p)$에서 γ를 구하는 Pollard의 Rho와 Brent의 이산대수 알고리즘의 수행횟수를 크게 감소시키는 알고리즘을 제안하였다. 제안된 방법은 Brent 방법으로 충돌을 검출하였다. 차이점은 $x_0 = 1$ 대신 $x_0 = \alpha\beta$을, $y = x_i, (i = 2^k)$ 대신 크기가 10인 Queue에 $y_j \leftarrow x_i, (i = 2^k, 1 \le j \le 10)$를 저장하는 방법을, β_γ 대신 $\beta = \beta_\gamma, \beta_{\gamma'}, \beta_{\gamma^{-1}}$의 충돌을 찾는 방법을 적용하였다. 제안된 Queue 적용법은 $x_0 = y_0 = 1$로 β_γ의 충돌을 검출하는 Pollard의 Rho 알고리즘의 수행횟수를 65.02%, $x_0 = 1$으로 β_γ의 충돌을 검출하는 Brent 알고리즘의 수행횟수를 47.80% 감소시켰다.

Abstract This paper proposes a discrete logarithm algorithm that largely reduces execution times of Pollard's Rho and Brent's algorithm in obtaining γ from $\alpha^{\gamma} \equiv \beta (\mathrm{mod}\, p)$. The proposed algorithm can be distinguished from the conventional Brent's algorithm by three major features: it sets an initial value as $x_0 = \alpha\beta$ in lieu of $x_0 = 1$; replaces $y = x_i, (i = 2^k)$ pointer with $y_j \leftarrow x_i, (i = 2^k, 1 \le j \le 10)$ for a Queue the size 10; and detects collision of $\beta_\gamma, \beta_{\gamma'}, \beta_{\gamma^{-1}}$ instead of β_γ. This Queue method has reduced the execution time of Pollard's Rho algorithm with $x_0 = y_0 = 1$ by 65.02%, and that of Brent's algorithm with $x_0 = 1$ by 47.80%.

Key Words : discrete logarithm, Pollard Rho algorithm, Brent Algorithm, queue, stack

Ⅰ. 서 론

암호는 대칭암호인 비밀키와 비대칭 암호인 공개키로 구분된다. 비대칭 암호의 공개키 n은 합성수 (composite number)로 2개 소수 p,q를 선택하여 $n = p \times q$로 쉽게 계산될 수 있으나 역으로, n으로부터 역으로 p,q를 구하는 소인수분해하기 어렵다는 수학의 난제에 기반하고 있다.[1-3] 대칭암호는 소수 (prime number)를 사용하며, $\alpha^{\gamma} \equiv \beta (\mathrm{mod}\, p)$에서 α,β,p가 주어졌을 때 비밀키 γ을 구하는 이산대수 (discrete logarithm)의 수학적 난제에 기반하고 있다.[1-3]

본 논문에서는 이산대수 알고리즘을 제안한다. 이산대수 알고리즘으로는 아기걸음-거인걸음 (Baby-step Giant-step), Pollard의 캥거루와 Rho (ρ), Pohlig- Hellman, Index calcurus, Number Field Sieves, Function Field Sieve 등이 있다.[2]

*정회원, 강릉원주대학교 과학기술대학 멀티미디어공학과
접수일자: 2017년 4월 13일, 수정완료: 2017년 5월 13일
게재확정일자: 2017년 6월 9일

Received: 13 April, 2017 / Revised: 13 May, 2017 /
Accepted: 9 June, 2017
*Corresponding Author: sulee@gwnu.ac.kr
Dept. of Multimedia Eng., Gangneung-Wonju National University, Korea

Shank의 아기걸음-거인걸음 알고리즘은 거인걸음 보폭 $m = \lceil \sqrt{n} \rceil$ 개의 데이터를 저장해야 하는 단점을 갖고 있으며, 수행 복잡도는 $O(\sqrt{n})$이다.[3,4] 반면에, Pollard의 Rho 알고리즘은 단지 2개의 포인터만 사용하여 충돌을 검출하는 방법으로 메모리 문제를 해결하였으나 수행 복잡도는 $O(\sqrt{n})$이다.[5] Pollard의 Rho 알고리즘에 대한 후속 연구가 Brent[6,7], Teske[8]와 Cheon et al.[9]에 의해 수행되었다. 지금까지는 Pollard의 Rho 알고리즘이 대칭키의 암호를 해독하는 이산대수 문제에 대해 가장 효율적인 방법으로 알려져 있다.

본 논문은 Pollard의 Rho와 Brent 알고리즘의 수행횟수를 크게 감소시키는 알고리즘을 제안한다. 2장에서는 사이클 검출 방법을 고찰한다. 3장에서는 Queue를 이용한 사이클 검출 방법을 제안하고, 성능을 검증하여 본다.

II. 이산대수의 사이클 검출

$\alpha^{\gamma} \equiv \beta \pmod p$에서 γ을 찾는 이산대수 문제에서 충돌 또는 합동 (collision or congruence)을 찾는 방법은 아기걸음-거인걸음[4], Pollard's Rho[5], Brent[6], Stack을 이용하는 방법[10,11] 등이 있다.

아기걸음-거인걸음법은 p를 $m = \lceil \sqrt{p} \rceil$ 보폭으로 m 걸음으로 분할하고, 아기걸음 단계에서는 첫 번째 거인보폭 m개에 대해 보폭 1인 아기걸음으로 $\alpha^i \pmod p, i = [0, m-1]$의 모듈러 지수연산을 수행한다. 다음으로 거인걸음을 역으로 걷기 위해 $\alpha^m \pmod p$의 역함수 $\alpha^{-m} \pmod p$을 유클리드 알고리즘으로 구한다. 마지막으로 거인걸음 단계에서는 보폭 $m = -m$으로 j번째 걸음에서 i번째 값과 일치하면, $\gamma = jm + i$로 결정한다.[3] 이 방법의 문제점은 아기걸음단계에서 구한 $\lceil \sqrt{p} \rceil$ 개의 모듈로 값을 계산 및 저장하고 탐색하는데 과다한 시간과 메모리를 필요로 한다는 점이다.[3] p가 큰 수이면 모듈로 계산과 저장 및 탐색에는 현실적으로 불가능하다. 이러한 데이터 저장 문제점을 해결한 알고리즘이 Pollard의 Rho 알고리즘[3,5]이다.

Pollard의 Rho 알고리즘[5]은 $\alpha^{\gamma} \equiv \beta \pmod n$에서 γ을 구하기 위해 식 (1)을 찾으며, $\alpha^a \beta^b \equiv \alpha^A \beta^B$의 충돌을 찾기 위해 $x_0 = y_0 = 1 (a = b = 0)$에서 거북이와 토끼가 동시에 출발하여 거북이는 정상 속도인 $x \leftarrow f(x)$로, 토끼는 거북이의 2배 속도인 $y \leftarrow f(f(y))$로 달려가면서 토끼와 거북이의 값이 같으면 종료하는 방식으로 수행된다. 여기서 x_i와 y_i는 식 (2)와 같이 계산된다. 이 방법은 사이클 검출에 있어서 가장 효율적인 방법으로 알려진 Floyd의 사이클 검출 알고리즘에 기반하고 있다.[11] 또한, 사이클을 검출하는 방법이 그리스 문자 Rho (ρ)와 유사하여 Rho 알고리즘이라 명명하였다.

$$\alpha^a \beta^b \equiv \alpha^A \beta^B, \ (B-b)\gamma = (a-A) \tag{1}$$

$$\alpha^a (\alpha^{\gamma})^b \equiv \alpha^A (\alpha^{\gamma}) \beta^B, \alpha^{a+\gamma b} \equiv \alpha^{A+\gamma B},$$

$$(a+\gamma b) = (A+\gamma B), \gamma(B-b) = (a-A)$$

$$f(x) = \begin{cases} x_{i-1}^2 \pmod p, & a_i = 2a_{i-1} \pmod G, b_i = 2b_{i-1} \pmod G \\ \alpha x_{i-1} \pmod p, & a_i = a_{i-1} + 1 \pmod G, b_i = b_{i-1} \\ \beta x_{i-1} \pmod p, & a_i = a_{i-1}, b_i = b_{i-1} + 1 \pmod G \end{cases}$$

$$f(y) = \begin{cases} y_{i-1}^2 \pmod p, & A_i = 2A_{i-1} \pmod G, B_i = 2B_{i-1} \pmod G \\ \alpha y_{i-1} \pmod p, & A_i = A_{i-1} + 1 \pmod G, B_i = B_{i-1} \\ \beta y_{i-1} \pmod p, & A_i = A_{i-1}, B_i = B_{i-1} + 1 \pmod G \end{cases} \tag{2}$$

Pollard의 Rho 알고리즘 일반형은 x_i와 X_i를 식 (3)으로 계산한다. 여기서 $M = \alpha^m, N = \beta^n$으로 m, n을 임의로 설정한다.[7,8]

$$f(x) = \begin{cases} x_{i-1}^2 \pmod p, & a_i = 2a_{i-1} \pmod G, b_i = 2b_{i-1} \pmod G \\ Mx_{i-1} \pmod p, & a_i = a_{i-1} + m \pmod G, b_i = b_{i-1} \\ Nx_{i-1} \pmod p, & a_i = a_{i-1}, b_i = b_{i-1} + n \pmod G \end{cases} \tag{3}$$

일반형의 대표적인 사례로 Teske[8]가 있다. Teske는 m, n 각각에 대해 선형 걸음 (linear walk)로 20회 곱셈을, 결합 걸음 (combined walk)로 16회 곱셈과 4회 제곱을 적용하였다.[3]

Brent 알고리즘[6]은 Pollard의 Rho 알고리즘에서 $y \leftarrow f(f(y))$를 적용하지 않고, 단지 $x \leftarrow f(x)$만 활용한다. 이 알고리즘은 $y \leftarrow x_i, (i = 2^k)$로 수행횟수 i가 2의 배수일 때 x_i의 값을 y에 저장하여 비교하는 방식이다. 이로 인해 Pollard의 Rho 알고리즘은 1회 수행에 $x \leftarrow f(x)$와 $y \leftarrow f(f(y))$의 3회 모듈러 연산을 수행하는 것을 1회로 감소시켜 Floyd의 사이클 검출 알고리즘에 비해 약 36% 빠르며, Pollard의 Rho 알고리즘의 수행시간을 24% 향상시켰다.[12]

Pollard와 Brent는 사이클 검출을 위해 2개의 포인터 (pointer)만을 활용하였다. Brent 알고리즘[6]의 경우 y를 1개만 적용함에 따라 이전 y값과 동일하여 수행횟수를 줄일 수 있음에도 불구하고 수행횟수가 많아지는 경향이 있다.

Brent 알고리즘의 이러한 문제점을 개선한 방법이 Nivasch[10]의 스택 (Stack)을 활용하는 방법이다. 이 방법은 Brent 알고리즘과 같이 $x \leftarrow f(x)$만을 적용하며, 임의의 i번째 수행횟수에서의 x_i 값이 기존에 스택에 저장된 값보다 작으면 스택에서 x_i보다 큰 값을 제거하며, 그렇지 않으면 스택에 추가하는 방법으로 x_i가 기존에 스택에 저장된 값과 동일하면 알고리즘을 종료한다. 즉, 스택에는 오름차순으로 값이 저장된다. 이 방법은 Brent 알고리즘에 비해 20% 빠른 것으로 증명되었다.[10] 그러나 스택 값을 갱신하는데 시간이 많이 소요될 수 있는 단점이 있다. Lee[3]는 Pollard의 Rho 알고리즘을 병렬로 수행하는 방법으로 수행시간을 단축하기도 하였다. Lee[3]은 Pollard Rho 기본형의 거북이는 정상 속도, 토끼는 거북이의 2배 속도로 걸어가면서 충돌 (합동)이 발생하는 지점을 찾는데 시간이 많이 소요되며, 이 지점까지 도달하기 이전에도 이 지점의 값과 동일한 값을 갖는 지점이 존재한다는 점에 착안하여 Pollard Rho 기본형을 다양하게 변형시킨 일반형 Pollard Rho 방법 4가지를 제안하였으며, 기본형 Pollard Rho 알고리즘에 비해 수행횟수를 71.70% 감소시켰다.

$2^{10} \equiv 5 (\mathrm{mod}\, 1019)$에서 $b=10$을 찾기 위해 아기걸음-거인걸음, Pollard의 Rho, Brent와 Nivasch의 Stack 이용법 수행 횟수를 비교하여 보자. 이 데이터의 사이클 시작점을 μ, 사이클 길이를 λ하 하면 $\mu=35, (x_{35}=974)$, $\lambda=17$이다. 따라서 $x_{35}=974=x_{52}$가 사이클 충돌로 최적의 수행횟수는 52회이다.

아기걸음-거인걸음은 $m=\lceil \sqrt{1019} \rceil = 32$로, 아기걸음 단계에서 $[0,31]$에 대해 $2^i \equiv c_i (\mathrm{mod}\, n), c_i = c_{i-1}^2 (\mathrm{mod}\, n)$을 구하는 과정에서 $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$을 찾아 거인걸음 단계를 수행하지 않고 10회 수행으로 해를 찾는다.

Pollard의 Rho 알고리즘은 Lee[3]에 따르면 51회 수행으로 $2^{681} 5^{378} = 1010 = 2^{301} 5^{416} (\mathrm{mod}\, 1019)$을 찾아 $(416-378)\gamma = (681-301) (\mathrm{mod}\, 1018), 38\gamma = 380, \gamma = 10$을 구한다. 즉, 모듈러 연산 횟수는 $51 \times 3 = 153$회이다.

Brent 알고리즘 수행 결과는 표 1에 제시되어 있으며, 81회 모듈러 연산을 수행하여 $x_{81}=640=x_{64}$를 찾았다. Nivasch의 Stack 이용법 수행 결과는 표 2에 제시되어 있으며, 62회를 수행하여 $x_{62}=86=x_{45}$를 찾았다. 여기서는 스택 값 갱신 횟수와 시간은 고려하지 않았다.

$2^{10} \equiv 5 (\mathrm{mod}\, 1019)$에서 $\gamma=10$을 찾는데 아기걸음-거인걸음 알고리즘의 수행횟수가 최소임을 알 수 있다.

표 1. $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$의 Brent 알고리즘 수행
Table 1. Brent's Algorithm for $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$

수행횟수	x_i	y	$x_i - y$	수행횟수	x_i	y	$x_i - y$
1	2	1	1	42	124	201	-77
2	10	1	9	43	248	201	47
3	20	10	10	44	221	201	20
4	100	10	90	45	86	201	-115
5	200	100	100	46	430	201	229
6	1000	100	900	47	860	201	659
7	981	100	881	48	224	201	23
8	425	100	325	49	101	201	-100
9	87	425	-338	50	505	201	304
10	436	425	11	51	1010	201	809
11	872	425	447	52	974	201	773
12	284	425	-141	53	794	201	593
13	401	425	-24	54	913	201	712
14	986	425	561	55	807	201	606
15	854	425	429	56	108	201	-93
16	194	425	-231	57	455	201	254
17	970	194	776	58	237	201	36
18	921	194	727	59	124	201	-77
19	433	194	239	60	248	201	47
20	866	194	672	61	221	201	20
21	254	194	60	62	86	201	-115
22	251	194	57	63	430	201	229
23	236	194	42	64	860	201	659
24	161	194	-33	65	224	860	-636
25	805	194	611	66	101	860	-759
26	591	194	397	67	505	860	-355
27	783	194	589	68	1010	860	150
28	670	194	476	69	974	860	114
29	321	194	127	70	794	860	-66
30	122	194	-72	71	913	860	53
31	610	194	416	72	807	860	-53
32	201	194	7	73	108	860	-752
33	660	201	459	74	455	860	-405
34	487	201	286	75	237	860	-623
35	974	201	773	76	124	860	-736
36	794	201	593	77	248	860	-612
37	913	201	712	78	221	860	-639
38	807	201	606	79	86	860	-774
39	108	201	-93	80	430	860	-430
40	455	201	254	81	**860**	**860**	**0**
41	237	201	36				

표 2. $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$의 Nivasch의 스택 알고리즘
Table 2. Nivasch's Stack Algorithm for $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$

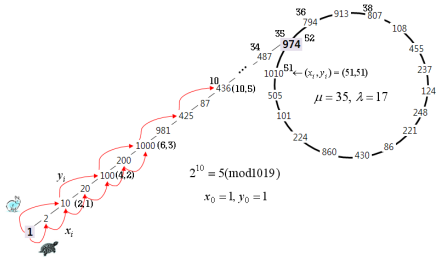
수행횟수	x_i	Stack 값	
		변경전	변경
1	2	-	2
2	10	2	2,10
3	20	2,10	2,10,20
4	100	2,10,20	2,10,20,100
5	200	2,10,20,100	2,10,20,100,200
6	1000	2,10,20,100,200	2,10,20,100,200,1000
7	981	2,10,20,100,200,1000	2,10,20,100,200,981
8	425	2,10,20,100,200,981	2,10,20,100,200,425
9	87	2,10,20,100,200,425	2,10,20,87
10	436	2,10,20,87	2,10,20,87,436
11	872	2,10,20,87,436	2,10,20,87,436,872
12	284	2,10,20,87,436,872	2,10,20,87,284
13	401	2,10,20,87,284	2,10,20,87,284,401
14	986	2,10,20,87,284,401	2,10,20,87,284,401,986
15	854	2,10,20,87,284,401,986	2,10,20,87,284,401,854
16	194	2,10,20,87,284,401,854	2,10,20,87,194
17	970	2,10,20,87,194	2,10,20,87,194,970
18	921	2,10,20,87,194,970	2,10,20,87,194,921
19	433	2,10,20,87,194,921	2,10,20,87,194,433
20	866	2,10,20,87,194,433	2,10,20,87,194,433,866
21	254	2,10,20,87,194,433,866	2,10,20,87,194,254
22	251	2,10,20,87,194,254	2,10,20,87,194,251
23	236	2,10,20,87,194,251	2,10,20,87,194,236

		2,10,20,87,194,236	2,10,20,87,161
24	161	2,10,20,87,161	2,10,20,87,161,805
25	805	2,10,20,87,161,805	2,10,20,87,161,591
26	591	2,10,20,87,161,591	2,10,20,87,161,591,783
27	783	2,10,20,87,161,591,783	2,10,20,87,161,591,670
28	670	2,10,20,87,161,591,670	2,10,20,87,161,321
29	321	2,10,20,87,161,321	2,10,20,87,122
30	122	2,10,20,87,122	2,10,20,87,122,610
31	610	2,10,20,87,122,610	2,10,20,87,122,201
32	201	2,10,20,87,122,201	2,10,20,87,122,201,660
33	660	2,10,20,87,122,201,660	2,10,20,87,122,201,487
34	487	2,10,20,87,122,201,487	2,10,20,87,122,201,487,974
35	974	2,10,20,87,122,201,487,974	2,10,20,87,122,201,487,794
36	794	2,10,20,87,122,201,487,794	
37	913		2,10,20,87,122,201,487,794,913
38	807	2,10,20,87,122,201,487,794,913	
39	108		2,10,20,87,122,201,487,794,807
40	455	2,10,20,87,122,201,487,794,807	2,10,20,87,108
41	237	2,10,20,87,108	2,10,20,87,108,455
42	124	2,10,20,87,108,455	2,10,20,87,108,237
		2,10,20,87,108,237	2,10,20,87,108,124

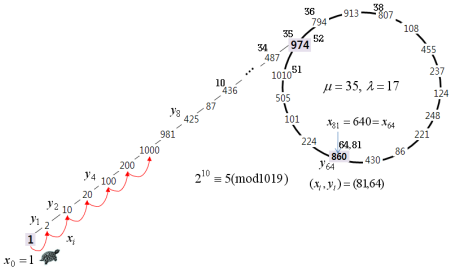
수행 횟수	x_i	Stack 값	
		변경전	변경
43	248	2,10,20,87,108,124	2,10,20,87,108,124,248
44	221	2,10,20,87,108,124,248	2,10,20,87,108,124,221
45	86	2,10,20,87,108,124,221	2,10,20,86
46	430	2,10,20,86	2,10,20,86,430
47	860	2,10,20,86,430	2,10,20,86,430,860
48	224	2,10,20,86,430,860	2,10,20,86,224
49	101	2,10,20,86,224	2,10,20,86,101
50	505	2,10,20,86,101	2,10,20,86,101,505
51	1010	2,10,20,86,101,505	2,10,20,86,101,505,1010
52	974	2,10,20,86,101,505,1010	2,10,20,86,101,505,974
53	794	2,10,20,86,101,505,974	2,10,20,86,101,505,794
54	913	2,10,20,86,101,505,794	2,10,20,86,101,505,794,913
55	807	2,10,20,86,101,505,794,913	2,10,20,86,101,505,794,913
56	108	2,10,20,86,101,505,794,913	2,10,20,86,101,108
57	455	2,10,20,86,101,108	2,10,20,86,101,108,455
58	237	2,10,20,86,101,108,455	2,10,20,86,101,108,237
59	124	2,10,20,86,101,108,237	2,10,20,86,101,108,124
60	248	2,10,20,86,101,108,124	2,10,20,86,101,108,124,248
61	221	2,10,20,86,101,108,124,248	2,10,20,86,101,108,124,221
62	**86**	2,10,20,**86**,101,108,124,221	2,10,20,**86**,**86**

그러나 이 알고리즘은 γ가 $\sqrt{n}$에서 멀어지고 p에 근접할수록 수행횟수는 점점 증가하며, 저장 메모리 문제가 발생한다. 따라서 본 논문에서는 아기걸음-거인걸음법은 고려하지 않는다.

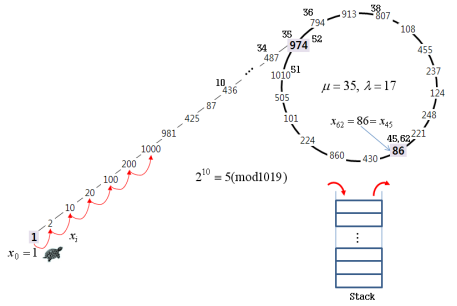
$2^{10} \equiv 5 (\mathrm{mod}\, 1019)$에서 $\gamma = 10$을 찾는 Pollard의 Rho, Brent와 Nivasch의 스택 이용법을 표현하면 그림 1과 같다.



(a) Pollard의 Rho 알고리즘



(b) Brent 알고리즘



(c) Nivasch의 스택 알고리즘

그림 1. $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$**의 충돌 검출방법**
Fig. 1. Collision Detection Methods for $2^{10} \equiv 5 (\mathrm{mod}\, 1019)$

Ⅲ. Queue 이용 알고리즘

Brent 알고리즘의 문제점을 개선한 Nivasch의 스택 이용법임을 고찰하였다. 그러나 이 방법은 스택의 값을 갱신하는데 많은 시간이 소요될 수 있다. Nivasch[10]는 $p = 5 \times 10^8$의 경우 스택 크기가 [44,53]임을 제시하였다.

본 장에서는 Brent 알고리즘의 문제점을 해결하면서, 메모리 용량을 최소화 할 수 있는 알고리즘을 제안한다. 제안된 알고리즘의 특징은 다음과 같다.

(1) 초기치는 $x_0 = 1$ 대신 $x_0 = \alpha\beta$을 적용하여 Pollard의 Rho 알고리즘의 $x \leftarrow f(x)$인 식 (2)를 계산한다.

이와 같이 하는 이유는 Pollard의 Rho 알고리즘은 초기치 $(x_0, y_0) = (1,1)$에 비해 $(x_0, y_0) = (\alpha\beta, \alpha\beta)$이 수행횟수가 보다 적어질 수 있기 때문이다. 제안된 알고리즘에는 식 (3)의 일반형을 적용할 수도 있으나 본 장에서는 식 (2)를 적용한다.

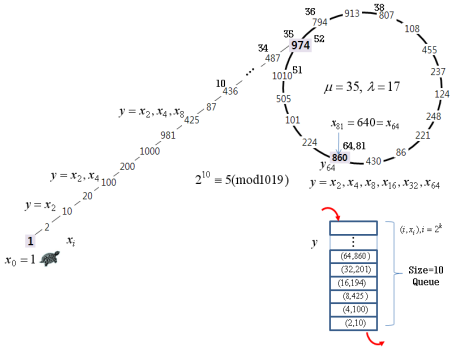
(2) 크기가 10인 Queue에 $y \leftarrow x_i, (i=2^k)$를 저장하는 방식을 적용한다. 이는 단일 y값만을 가지는 Brent 알고리즘과 차이가 있다.

만약, $i=2^k (k \geq 1)$이면 (i, x_i)를 Queue에 저장하며, Queue의 용량을 초과하면 $k-10$번째 값을 제거하고, k번째 값을 저장한다. 이와 같이 Queue를 사용하면 스택 사용법에 비해 메모리에 저장된 데이터 갱신 시간을 크게 단축시킬 수 있다.

(3) $\alpha^{\gamma} \equiv \beta_{\gamma} \pmod p$에 대해 $\beta_{\gamma}, \beta_{\gamma'}, \beta_{\gamma^{-1}}$을 적용한다. 즉, $\alpha^{\gamma'} \equiv \beta_{\gamma'} \pmod p$와 $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}} \pmod p$도 함께 적용한다.

주어진 수 p가 소수라면, $\alpha < p$에 대해 $\alpha^{(p-1)/2} \equiv \pm 1 \pmod p$이다. 만약, $\alpha^{(p-1)/2} \equiv -1$이면 $\gamma' = (p-1)/2 + \gamma \pmod{p-1}$에 대해 $\beta_{\gamma} + \beta_{\gamma'} = p$가 존재한다. 그러나 $\alpha^{(p-1)/2} \equiv 1$이면 $\beta_{\gamma} = \beta_{\gamma'}$이다. 또한, $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}} \pmod p$은 $\alpha^{\gamma} \equiv \beta_{\gamma} \pmod p$의 역함수로 $\gamma^{-1} = (p-1) - \gamma$이다. 만약, $\beta_{\gamma'}$의 충돌을 검출하여 γ'를 구하면 $\gamma = \gamma' - (p-1)/2$로, $\beta_{\gamma^{-1}}$의 충돌을 검출하여 γ^{-1}을 구하면 $\gamma = (p-1) - \gamma^{-1}$로 구할 수 있다.

Queue를 이용하는 제안된 알고리즘으로 $x_0 = 1$과 $x_0 = \alpha\beta$인 경우 β_{γ}을 구하는 방법은 그림 2에 제시되어 있다.



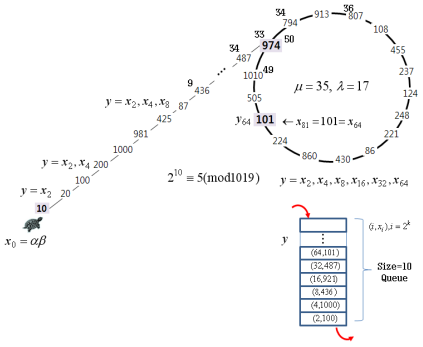


그림 2. Queue 이용 충돌 검출방법
Fig. 2. Collision Detection Methods Using a Queue

제안된 알고리즘을 $p=1019$, $\alpha=2,5$에 대해 충돌 검출 모듈러 연산횟수를 비교하여 본다. 실험에는 $\gamma=5,100,200,300,\cdots,1000$의 11개 값을 구하여 표 3에 제시하였다. Pollard의 Rho 알고리즘 수행횟수에서 xx(xx)의 ()는 수행횟수의 3배를 한 것이다.

첫 번째로, Pollard의 Rho 알고리즘의 β_{γ} 충동검출 횟수에서 $x_0 = y_0 = 1$과 $x_0 = y_0 = \alpha\beta$을 비교해 보면 $2^{10}, 5^{100}, 5^{400}, 5^{900}$에서 보다 적은 수행횟수를 보이며, 나머지는 동일한 수행횟수를 보여 6.57%의 수행회수 감소를 나타내고 있다. 이러한 현상은 Brent와 Queue를 적용할 때도 마찬가지 결과를 나타낸다. 따라서 $x_0 = y_0 = 1$ 대신 $x_0 = y_0 = \alpha\beta$를 적용한다.

두 번째로, β_{γ}의 충돌을 검출하는 $x_0 = y_0 = 1$인 Pollard Rho와 $x_0 = 1$인 Brent 알고리즘에 대해 $x_0 = \alpha\beta$, $\beta_{\gamma}, \beta_{\gamma'}, \beta_{\gamma^{-1}}$의 충돌을 검출하는 제안된 Queue 적용법과 수행횟수를 비교한 결과는 표 4에 제시되어 있다. 제안된 Queue 적용법은 Pollard의 Rho 알고리즘의 수행횟수를 65.02%, Brent 알고리즘의 수행횟수를 47.80% 감소시켰다. Lee[3]의 Pollard의 Rho 변형 알고리즘 4종을 병렬수행하는 방식과 비교시 13개와 10개에서 보다 빠른 결과를 얻어 약간 좋지 않은 결과를 얻었지만, 본 논문 방법은 단일 알고리즘 수행방식으로 메모리와 처리속도 측면에서는 보다 빠른 측면의 장점을 갖고 있다고 할 수 있다.

Ⅳ. 결론

본 논문은 이산대수를 계산하는 Pollard의 Rho와 Brent 알고리즘의 수행횟수를 크게 감소시킨 Queue를

표 3. $n=1019$의 충돌 검출 수행횟수

Table 3. The number of collision detection for $n=1019$

α	충돌 검출 횟수 $(\beta_\gamma,\beta_{\gamma'},\beta_{\gamma^{-1}})$																	
	$\alpha^\gamma=\beta_\gamma$	Pollard		Brent		Lee[3]	Queue		$\alpha^{\gamma'}=\beta_{\gamma'}$	Pollard	Brent	Queue	$\alpha^{\gamma^{-1}}=\beta_{\gamma^{-1}}$	Pollard	Brent	Queue
		1	$\alpha\beta$	1	$\alpha\beta$	$(\beta_\gamma,\beta_{\gamma'},\beta_{\gamma^{-1}})$	1	$\alpha\beta$		$\alpha\beta$	$\alpha\beta$	$\alpha\beta$		$\alpha\beta$	$\alpha\beta$	$\alpha\beta$
2	$2^{10}\equiv 5$	51(153)	**34**	81	81	13x4=52	81	81	$2^{519}\equiv 1014$	57	83	83	$2^{1008}\equiv 204$	54	91	91
	$2^{100}\equiv 548$	19(57)	19	51	51	12x4=48	23	**20**	$2^{609}\equiv 471$	17	49	**33**	$2^{918}\equiv 225$	40	104	**56**
	$2^{200}\equiv 718$	60(180)	60	124	124	10x4=40	76	**68**	$2^{709}\equiv 301$	34	49	49	$2^{818}\equiv 694$	37	101	**53**
	$2^{300}\equiv 130$	20(60)	20	52	52	7x4=28	36	**28**	$2^{809}\equiv 889$	63	127	**64**	$2^{718}\equiv 243$	82	105	105
	$2^{400}\equiv 929$	79(237)	79	186	186	10x4=40	95	95	$2^{909}\equiv 90$	74	101	101	$2^{618}\equiv 668$	40	42	42
	$2^{500}\equiv 611$	36(108)	36	76	76	15x4=60	76	76	$2^{1009}\equiv 408$	36	100	**40**	$2^{518}\equiv 507$	74	101	101
	$2^{600}\equiv 596$	33(99)	33	97	97	8x4=32	41	**34**	$2^{91}\equiv 423$	76	102	102	$2^{418}\equiv 966$	45	73	73
	$2^{700}\equiv 528$	46(138)	46	110	110	10x4=40	78	78	$2^{191}\equiv 491$	31	63	63	$2^{318}\equiv 303$	46	87	87
	$2^{800}\equiv 967$	108(324)	108	307	307	17x4=68	182	182	$2^{291}\equiv 52$	41	105	**45**	$2^{218}\equiv 921$	84	274	**149**
	$2^{900}\equiv 36$	20(60)	20	26	26	6x4=24	26	26	$2^{391}\equiv 983$	30	47	47	$2^{118}\equiv 368$	60	76	76
	$2^{1000}\equiv 367$	70(210)	70	99	99	9x4=36	99	99	$2^{491}\equiv 652$	72	262	**137**	$2^{18}\equiv 261$	28	60	60
5	$5^{10}\equiv 548$	60(180)	60	124	124	4x4=16	92	92	-	-	-	-	$5^{1008}\equiv 225$	25	57	57
	$5^{100}\equiv 367$	30(90)	**15**	47	**31**	6x4=24	47	**31**	-	-	-	-	$5^{918}\equiv 261$	44	54	54
	$5^{200}\equiv 181$	28(84)	28	39	39	6x4=24	39	39	-	-	-	-	$5^{818}\equiv 867$	55	75	75
	$5^{300}\equiv 192$	19(57)	**19**	51	51	14x4=56	35	35	-	-	-	-	$5^{718}\equiv 69$	36	38	**38**
	$5^{400}\equiv 153$	18 (54)	**16**	34	**18**	9x4=36	34	**18**	-	-	-	-	$5^{618}\equiv 686$	9	25	**11**
	$5^{500}\equiv 106$	20(60)	20	52	52	3x4=12	21	21	-	-	-	-	$5^{518}\equiv 721$	40	104	**72**
	$5^{600}\equiv 1017$	78(234)	78	266	266	10x4=40	141	141	-	-	-	-	$5^{418}\equiv 685$	35	99	**37**
	$5^{700}\equiv 844$	27(81)	27	59	59	8x4=32	35	**31**	-	-	-	-	$5^{318}\equiv 460$	11	27	**12**
	$5^{800}\equiv 991$	38(114)	38	102	102	8x4=32	46	46	-	-	-	-	$5^{218}\equiv 837$	40	84	84
	$5^{900}\equiv 933$	50(150)	**25**	57	57	8x4=32	41	**23**	-	-	-	-	$5^{118}\equiv 391$	15	31	31
	$5^{1000}\equiv 27$	14(42)	14	30	30	8x4=32	30	30	-	-	-	-	$5^{18}\equiv 151$	48	112	**80**

표 4. 알고리즘 성능 비교

Table 4. Compare of algorithm performance

α	$\alpha^\gamma=\beta_\gamma$	β_γ			Lee[3]	Queue 이용 방법 $(x_0=\alpha\beta)$				Queue 이용 방법 수행횟수 비율	
		Pollard	Brent								
		$x_0=y_0=1$	$x_0=1$	비율	$(\beta_\gamma,\beta_{\gamma'},\beta_{\gamma^{-1}})$ 4종 병렬수행	β_γ,	$\beta_{\gamma'}$	$\beta_{\gamma^{-1}}$	최소값	Pollard Rho 알고리즘 대비 비율	Brent 알고리즘 대비 비율
2	$2^{10}\equiv 5$	51(153)	**81**	52.94 %	13x4=**52**	**81**	83	91	81	52.94 %	100.00 %
	$2^{100}\equiv 548$	19(57)	51	89.47 %	12x4=48	**20**	33	56	**20**	35.09 %	39.22 %
	$2^{200}\equiv 718$	60(180)	124	68.89 %	10x4=**40**	68	**49**	53	49	27.22 %	39.52 %
	$2^{300}\equiv 130$	20(60)	52	86.67 %	7x4=**28**	**28**	64	105	**28**	46.67 %	53.85 %
	$2^{400}\equiv 929$	79(237)	186	78.48 %	10x4=**40**	95	101	**42**	42	17.72 %	22.58 %
	$2^{500}\equiv 611$	36(108)	76	70.37 %	15x4=60	76	**40**	101	**40**	37.04 %	52.63 %
	$2^{600}\equiv 596$	33(99)	97	97.98 %	8x4=**32**	**34**	102	73	34	34.34 %	35.05 %
	$2^{700}\equiv 528$	46(138)	110	79.71 %	10x4=**40**	78	**63**	87	63	45.65 %	57.27 %
	$2^{800}\equiv 967$	108(324)	307	94.75 %	17x4=68	182	**45**	149	**45**	13.89 %	14.66 %
	$2^{900}\equiv 36$	20(60)	**26**	43.33 %	6x4=**24**	**26**	47	76	26	43.33 %	100.00 %
	$2^{1000}\equiv 367$	70(210)	99	47.14 %	9x4=**36**	99	137	**60**	60	28.57 %	60.61 %
5	$5^{10}\equiv 548$	60(180)	124	68.89 %	4x4=**16**	92	-	**57**	57	31.67 %	45.97 %
	$5^{100}\equiv 367$	30(90)	47	34.44 %	6x4=**24**	**31**	-	54	31	34.44 %	65.96 %
	$5^{200}\equiv 181$	28(84)	**39**	46.43 %	6x4=**24**	**39**	-	75	39	46.43 %	100.00 %
	$5^{300}\equiv 192$	19(57)	51	89.47 %	14x4=56	**35**	-	38	**35**	61.40 %	68.63 %
	$5^{400}\equiv 153$	18 (54)	34	33.33 %	9x4=36	18	-	**11**	**11**	20.37 %	32.35 %
	$5^{500}\equiv 106$	20(60)	52	86.67 %	3x4=**12**	**21**	-	72	21	35.00 %	40.38 %
	$5^{600}\equiv 1017$	78(234)	266	28.21 %	10x4=40	141	-	**37**	**37**	15.81 %	13.91 %
	$5^{700}\equiv 844$	27(81)	59	72.84 %	8x4=32	31	-	**12**	**12**	14.81 %	20.34 %
	$5^{800}\equiv 991$	38(114)	102	89.47 %	8x4=**32**	**46**	-	84	46	40.35 %	45.10 %
	$5^{900}\equiv 933$	50(150)	57	38.00 %	8x4=32	**23**	-	31	**23**	15.33 %	40.35 %
	$5^{1000}\equiv 27$	14(42)	**30**	71.43 %	8x4=32	**30**	-	80	**30**	71.43 %	100.00 %
평균	수행비율	-		**66.77 %**	-	-				**34.98 %**	**52.20 %**
	감소율	-		**33.23 %**	-	-				**65.02 %**	**47.80 %**

적용하는 방법을 제안하였다.

제안된 방법은 $\beta_\gamma, \beta_{\gamma'}, \beta_{\gamma^{-1}}$의 충돌을 검출하기 위해 $x_0=\alpha\beta$과 크기가 10인 Queue에 $y_j \leftarrow x_i, (i=2^k, 1\le j \le 10)$를 저장하는 방법을 적용하였다. 제안된 Queue 적용법은 $x_0=y_0=1$으로 β_γ의 충돌을 검출하는 Pollard의 Rho 알고리즘의 수행횟수를 65.02%, $x_0=1, y=x_i, (i=2^k)$로

β_γ의 충돌을 검출하는 Brent 알고리즘의 수행횟수를 47.80% 감소시켰다.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, Section 31.7 The RSA Public-key Cryptosystem", 2nd Ed., MIT Press and McGraw-Hill, ISBN: 0-262-03293-7, pp. 881 - 887, 2001.

[2] D. R. Stinson, "Cryptography: Theory and Practice," 3rd ed., London, CRC Press, ISBN-10: 1584885084, 2006.

[3] S. U. Lee, "Multiple Parallel-Pollard's Rho Discrete Logarithm Algorithm," Journal of KSCI, Vol. 20, No. 8, pp. 29-33, Aug. 2015.
DOI: https://doi.org/10.9708/jksci.2015.20.8.029

[4] D. Shanks, "The Infrastructure of a Real Quadratic Field and its Applications", Proceedings of the 1972 Number Theory Conference, University of Colorado, Boulder, pp. 217-224, MR 389842, 1972.

[5] J. M. Pollard, "Monte Carlo Methods for Index Computation (mod p)," Mathematics of Computation, Vol. 32, No. 143, pp. 918-924, Jul. 1978.
DOI: https://doi.org/10.2307/2006496

[6] R. P. Brent, "An Improved Monte Carlo Factorization Algorithm," Bit Numerical Mathematics (BIT), Vol. 20, No. 2, pp. 176-184, Jun. 1980.
DOI: https://doi.org/10.1007/BF01933190

[7] S. Bai and R. P. Brent, "On the Efficiency of Pollard's Rho Method for Discrete Logarithms," Computing: The Australasian Theory Symposium (CATS), Vol. 77, pp. 125-131, 2008.

[8] E. Teske, "Speeding Up Pollard's Rho Method for Computing Discrete Logarithms," Lecture Notes in Computer Science, Vol. 1423, pp. 541-554, Jun. 1998.
DOI: https://doi.org/10.1007/BFb0054891

[9] J. H. Cheon, J. Hong, and M. K. Kim, "Speeding Up the Pollard Rho Method on Finite Fields," ASIACRYPT, pp. 471-488, 2008.

[10] G. Nivasch, "Cycle Detection Using a Stack," Information Processing Letters, Vol. 90, No. 3, pp. 135-140, May 2004.
DOI: https://doi.org/10.1016/j.ipl.2004.01.016

[11] A. Shamir, "Random Graphs in Cryptography," 7th Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms, 2007.

[12] Wikipedia, "Cycle Detection", http://en. wikipedia. org/wiki/Cycle_detection, Wikimedia Foundation, Inc, 2015.

저자 소개

이 상 운(정회원)

- 1987년: 한국항공대학교 항공전자공학과 (학사)
- 1997년: 경상대학교 컴퓨터과학과 (석사)
- 2001년 : 경상대학교 컴퓨터과학과 (박사)
- 2003년 : 강원도립대학 컴퓨터응용과 전임강사
- 2004년~2007.2 : 국립 원주대학 여성교양과 조교수
- 2007.3~2015.3 : 강릉원주대학교 멀티미디어공학과 부교수
- 2015.4~현재 : 강릉원주대학교 멀티미디어공학과 정교수

<주관심분야 :소프트웨어 프로젝트 관리, 개발 방법론, 분석과 설계 방법론, 시험 및 품질보증, 소프트웨어 신뢰성, 그래프 알고리즘>

- e-mail : sulee@gwnu.ac.kr